Before you start:

Homework Files

You can download the starter files for coding as well as this *tex* file (you only need to modify *homework4.tex*) on canvas and do your homework with latex. Or you can scan your handwriting, convert to pdf file, and upload it to canvas before the due date. If you choose to write down your answers by hand, you can directly download the pdf file on canvas which provides more blank space for solution box.

Submission Form

For homework 4, there are only one part of submission, which is a pdf file as your solution named as VE281_HW4_[Your Student ID]_[Your name].pdf uploaded to canvas.
Estimated time used for this homework: **4-5 hours.**

0 Student Info (0 point)

Your name and student id:

Solution: WANGZINING 520370910042

1 Graph Search MCQs (16 points)

Choose the right answer of the following multiple choice questions, no explanation is needed.

1. Suppose that we have an undirected graph $G = (V, E)$, where

   - $V = \{a, b, c, d, e, f\}$
   - $E = \{(a, b), (a, e), (a, c), (b, e), (c, f), (f, d), (e, d)\}$

   If we perform a DFS from node a, which of the following could be a possible node sequence?

   A. a, b, e, c, d, f

   B. a, c, f, e, b ,d

   C. a, e, b, c, f ,d

   D. a, e, d, f, c, b

   Solution: D

2. After learning about the concept of graph, you can know that tree can be represented by an acyclic directed graph. Comparing tree with such representation and other kinds of directed graph, which of the following is right?

   A. A node in **a tree** could have multiple parents

   B. A node in **a directed graph** could have multiple predecessors

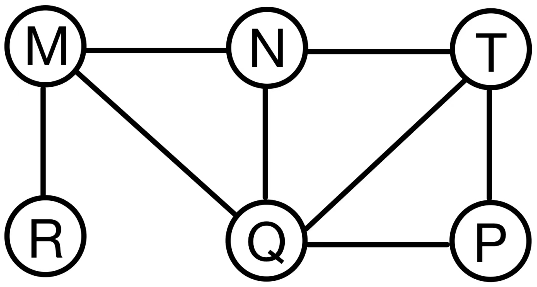
   C. A node in **a tree** could **NOT** have multiple children

   D. A node in **a directed graph** could **NOT** have multiple successors

   Hint: If there exists an edge from node u to node v in a directed graph, we then call u the predecessor of v, and call v the successor of u.

   Solution: B

3. The Breadth First Search (BFS) algorithm has been implemented using the queue data structure. Which one of the following is a possible order of visiting the nodes in the graph below?



A. MNTPQR

B. NQMPTR

C. QMNRTP

D. PTQNMR

Solution: D

4. Let G be a directed graph whose vertex set is the set of numbers from 1 to 100. There is an edge from a vertex i to a vertex j if and only if either $j = i + 1$ or $j = 3i$. The minimum number of edges in a path in G from vertex 1 to vertex 100 is

   A. 23

   B. 99

   C. 4

   D. 7

Solution: D

2 Topological Sort (8 points)

In the lecture, we have introduced how to implement topological sorting algorithm by queue, whose main idea is quite similar to BFS. Mr. Blue Tiger argues that without **in-degree**, we can also implement topological sorting by using DFS, with an array **visited**. Complete the following pseudo code provided by him:

Algorithm 1 Algorithm to implement topological sorting with DFS

```
Input: an adjacency list representing the graph
Output: a stack with the topologically sorted nodes.
Create a stack $S$ and a boolean array $visited[]$ initialized with false.
for each node $v$ in the graph do
    dfs_helper($v$, $visited$, $S$)
end for

dfs_helper($v$, $visited$, $S$):
$visited[v] = true$

for each node $u$ adjacent to $v$ do
    if ($visited[u]$) return;
    dfs_helper($u$,$visited$,$S$)

end for
S.push(v)
```

3 Minimum Spanning Tree (12 points)

3.1 Delete an edge (6 points)

Given a minimum spanning tree of a connected undirected graph G, an edge is deleted from G now. Suppose after deletion, G is still connected. Describe how to find the new minimum spanning tree with the old one in O(E) time.

Solution: If the deleted edge is not the one in the tree, the tree have no changes.
The new tree is the same as the old tree.
If the deleted edge is in the tree, store the two node as a and b and delete the edge in the tree.
Use dfs on a and b, for any node v that can be visited from a, set $flag[v]$ as 0.
For any node u that can be visited from b, set $flag[u]$ as 1.
$e_f = 0$.
$w_f = INF$.
For each edge e in G, denote the two node of e as x and y, if ((flag[x]+flag[y]==1) and $(weight[e] < w_f)$), $w_f = weight[e], e_f = e$.
Add e_f to the tree and that is the new tree.

3.2 Add an edge (6 points)

Given a minimum spanning tree of a connected undirected graph G, an edge is added to G now. Describe how to find the new minimum spanning tree with the old one in O(V) time.

Solution: Denote the added edge as e_i, and the two node it connected as a and b.
Use BFS to find path from a to b, store the path in $p[]$.
Find the edge with greatest weight among $p[] + e_i$, denote as e_g.
If the $e_g == e_i$, the tree have no change. Directly return.

Remove e_g and insert e_i into the tree.
Return.

4 Shortest Path (4 points)

As introduced in the lecture, we can use Dijkstra's algorithm when the graph only has non-negative edges. Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Then briefly explain why Dijkstra's algorithm fails on your example.

Solution:
d(a,b)=20; d(b,c)=20; d(c,d)=20; d(d,e)=20; d(e,f)=-10080; d(a,f)=5;
Use Dijkstra we will get the shortest path from a to f is 5, but the correct answer is -10000.
Because in Dijkstra we have assumed all path is non-negative,so we have $d(a,b)+d(b,c) \geq d(a,b)$.
So in the example, we have $d(a,b) + d(b,c) + d(c,d) + d(e,f) \geq d(a,b) \geq d(a,f)$, so we will not take the real shortest path into account.
Hence, we get a incorrect answer.

5 Dynamic Programming (60 points)

5.1 Basic Case (12 points)

Suppose that we have an $n * n$ matrix filled with integers. Starting from the top left corner, we advance to either the downward or rightward block for each step and finally reach the bottom right corner of the matrix. During this process, we will pass through nodes with different integers. By applying dynamic programming, we can find the path with the largest sum of the passed integers. Write out the recurrence relation.

Solution: dp[i][j] = max(dp[i-1][j],dp[i][j-1])+integers[i][j] (Assume from [1][1] to [n][n]) and set dp[0][j] and dp[i][0] be -INF. Set dp[1][1] be integers[1][1])

5.2 Do it twice? (24 points)

Previously, we just go across the matrix for a single time. Assume that after the first travel, we set the visited integer in the matrix to be 0 and go across the matrix, back to the left corner again. How to maximize the sum of the passed integers in the whole procedure (top left $->$ bottom right $->$ top left)? In terms of this problem, Alice and Bob have different ideas again.

5.2.1 Simple repeat (10 points)

Bob thinks that since this problem is quite similar to what we have solved in Section 5.1, just run dynamic programming twice and add them up, and we will have the correct final result. Do you agree with him? If agree, write the recurrence relation for the second dynamic programming procedure and state whether there is a difference; if not agree, come up with a counter example and explain why it doesn't work.

Solution:
Disagree.
0 6 6 3 9 0 0 3 0
The first one is $0->6->9->3->0$.
The second one is $0->0->6->0->0$.
The left 3 is missed.
However, the best way is $0->3->9->3->0$ and $0->0->6->6->0$, which can take all numbers.
It can not apply greedy algorithm.

5.2.2 Double the result table (14 points)

Alice thinks that the dynamic programming strategy for this problem should be modified from the very beginning in this case. She gives out the main function as shown below. However, after testing, she found out there are some problems within this piece of code. Please correct the code between line 17 to line 26.

```cpp
int main(){
    // two paths are considered simultaneously, one at (i, j), the other at (k, l)
    // integers stored in integers[n][n]
    int n;
    cin >> n;
    int dp[100][100][100][100] = {0};
    int integers[100][100] = {0};
    // read all the inputs
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            cin >> integers[i][j];

    dp[0][0][0][0] = integers[0][0]; // start point

    // start dp

    /* modify code within this part */
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            for (int k = 0; k < n; k++)
                for (int l = 0; l < n; l++)
                {
                    dp[i][j][k][l] = max(dp[i-1][j][k-1][l], dp[i][j-1][k-1][l],
                                         dp[i-1][j][k][l-1], dp[i][j-1][k][l-1])
                                     +integers[i][j]+integers[k][l];
                    // it will be 0 after first visit.
                    if ( (i==k) && (j==l) )
                    {
                        dp[i][j][k][l] -=integers[i][j];
                    }
                }

    /* modify code within this part */

    cout << dp[n][n][n][n];

}
```

5.3 How to save memory? (12 points)

Bob takes a look at Alice's strategy and thinks that its memory usage is too bad. Regardless of correctness, it will have a space complexity of $O(n^4)$, which sounds horrible. Propose a modification to this dynamic programming algorithm so that the space complexity can be reduced to $O(n^3)$ and write out its recurrence relation.
Hint: one way to do so is to reduce the array *dp* to be 3-dimensional. There should be 1 dimension still for i and 1 dimension still for k.

Solution: First, there is no difference whether the second visit is from right bottom or from left top.
So we assume both visit from the same point (the left top).
Each iteration can go for both visit.
So we can set $dp[i][k][l]$, i and k have same meaning and l means how long two visits move, the y-axis can be calculated by (l and i) and (l and j).
We have

$dp[i][k][l] = max(dp[i-1][k][l-1], dp[i][k-1][l-1], dp[i-1][k-1][l-1], dp[i][k][l-1]) + integers[i][l+2-i] + integers[k][l+2-k]$ if $(i! = k)$.
$dp[i][k][l] = max(dp[i-1][k][l-1], dp[i][k-1][l-1], dp[i-1][k-1][l-1], dp[i][k][l-1]) + integers[i][l+2-i]$ if $(i == k)$.

5.4 Does optimization end here? (12 points)

Looking at the modification proposed by Bob, Alice surprisingly agrees with him. After brainstorming, they find that this strategy can be further optimized in terms of space complexity. Briefly state how you can further reduce its space complexity to $O(2n^2)$.
Hint: Do we need every dp value in every iteration?

Solution:
Note that we only have change from $l-1$ to l, so storing this dim is not necessary. We can remove it and get two arrays, one for the previous result and one for the new result.
We can use div 2 to change between two arrays.
That is, we use dp[i][k][l%2] by having dp[n+1][n+1][2].
$O(2(n+1)(n+1)) = O(2n^2)$

Notes: Actually we can further reduce its space usage from $O(2n^2)$ to $O(n^2)$ in this problem.

Reference

Assignment 4, VE281, FA2021, UMJI-SJTU
Assignment 5, VE281, FA2021, UMJI-SJTU
Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne, Princeton University